Internalizing inductive-inductive definitions in Martin-Löf Type Theory

Fredrik Nordvall Forsberg

Swansea University

csfnf@swansea.ac.uk

München 27.11.2012

Joint work with Anton Setzer.
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system.
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).

- So I try.
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).

- So I try.

```plaintext
data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> "a \leq L ℓ" -> SList
```
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).

- So I try.

- The empty list is sorted.

```
data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> "a \leq L ℓ" -> SList
```
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).

- So I try.

- The empty list is sorted.

```haskell
data SList : Set where
  [] : SList
```
A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).

- So I try.

- The empty list is sorted.

- If I have a sorted list \(\ell = [\ell_0, \ldots, \ell_m]\), and an element \(a\), and \(a \leq \text{all } \ell_k \text{ in } \ell\), then \([a, \ell_0, \ldots, \ell_m]\) is a sorted list.

```haskell
data SList : Set where
  [] : SList
```

A data type of sorted lists?

- Say that I’m working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set \((A, \leq)\)).

- So I try.

- The empty list is sorted.

- If I have a sorted list \(\ell = [\ell_0, \ldots, \ell_m]\), and an element \(a\), and \(a \leq \text{all } \ell_k \text{ in } \ell\), then \([a, \ell_0, \ldots, \ell_m]\) is a sorted list.

```haskell
data SList : Set where
  [] : SList
  cons : (a : A) -> (\ell : SList) -> "a \leq_\ell \ell" -> SList
```
What is $\leq_L$?

```haskell
data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> "a $\leq_L\, \ell" \rightarrow SList

"a $\leq_L\, \ell"$ if $a \leq$ all elements of $\ell$.
```
What is $\leq_L$?

data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> "a $\leq_L$ ℓ" -> SList

- "a $\leq_L$ ℓ" if $a \leq$ all elements of ℓ.
- Informal? No! We want to express the specification in the types.
What is $\leq_L$?

data SList : Set where
  [] : SList
  cons : (a : A) → (ℓ : SList) → "a $\leq_L$ ℓ" → SList

• “$a \leq_L ℓ$” if $a \leq$ all elements of ℓ.

• Informal? No! We want to express the specification in the types.

• Natural inductive definition:
What is \(\leq_L\)?

data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> "a \(\leq_L\) ℓ" -> SList

- \(a \leq_L ℓ\) if \(a \leq\) all elements of \(ℓ\).
- Informal? No! We want to express the specification in the types.
- Natural inductive definition:
  - Every \(a\) is trivially smaller than all elements of the empty list \([]\).
What is $\leq_L$?

data SList : Set where
    [] : SList
    cons : (a : A) -> (ℓ : SList) -> "a $\leq_L$ ℓ" -> SList

- "$a \leq_L ℓ$" if $a \leq$ all elements of ℓ.

Informal? No! We want to express the specification in the types.

Natural inductive definition:

- Every $a$ is trivially smaller than all elements of the empty list [].
- If $x \leq a$ and inductively $x \leq_L ℓ$, then $x \leq_L \text{cons}(a, ℓ, p)$.
Sorted lists and $\leq_L$

```
data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> a $\leq_L$ ℓ -> SList

"a $\leq_L$ ℓ" if a $\leq$ all elements of ℓ.

Informal? No! We want to express the specification in the types.

Natural inductive definition:

  Every a is trivially smaller than all elements of the empty list [].

  If $x \leq a$ and inductively $x \leq_L ℓ$, then $x \leq_L$ cons(a, ℓ, p).
```

```
data $\leq_{L-} : \mathbb{N} ->$ SList $->$ Set where
  triv : $\forall a -> a$ $\leq_L$ []
  $\leq_{L-}$-cons : $\forall x$ $->$ x $\leq a$ $->$ x $\leq_L$ ℓ $->$ x $\leq_L$ cons(a, ℓ, p)
```
Sorted lists and $\leq_L$

data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> a $\leq_L$ ℓ -> SList

data $\leq_L : \mathbb{N} \rightarrow SList \rightarrow Set$ where
  triv : ∀ a -> a $\leq_L$ []
  $\leq_L$-cons : ∀ x -> x $\leq$ a -> x $\leq_L$ ℓ -> x $\leq_L$ cons(a, ℓ, p)
Sorted lists and $\leq_L$

mutual

data SList : Set where
  [] : SList
  cons : (a : A) -> (ℓ : SList) -> a $\leq_L$ ℓ -> SList

data $\leq_L$- : ℕ -> SList -> Set where
  triv : ∀ a -> a $\leq_L$ []
  $\leq_L$-cons : ∀ x -> x $\leq$ a -> x $\leq_L$ ℓ -> x $\leq_L$ cons(a, ℓ, p)

- Needs to be a mutual definition – cons refers to $\leq_L$, which is indexed by SList.
Sorted lists and \( \leq_L \)

mutual
data SList : Set where
[] : SList
cons : (a : A) -> (\ell : SList) -> a \leq_L \ell -> SList

data \( \leq_L \) : \( \mathbb{N} \) -> SList -> Set where
triv : \( \forall a \to a \leq_L [] \)
\( \leq_L \)-cons : \( \forall x \to x \leq a \to x \leq_L \ell \to x \leq_L \text{cons}(a, \ell, p) \)

• Needs to be a mutual definition – cons refers to \( \leq_L \), which is indexed by SList.

• Both SList and \( \leq_L \) defined inductively – an inductive-inductive definition!
Sorted lists and $\leq_L$

mutual
data SList : Set where

  [] : SList
  cons : (a : A) -> (ℓ : SList) -> a $\leq_L$ ℓ -> SList

data $\leq_L$ : ℕ -> SList -> Set where

  triv : ∀ a -> a $\leq_L$ []
  $\leq_L$-cons : ∀ x a ℓ p -> x $\leq_a$ a -> x $\leq_L$ ℓ -> x $\leq_L$ cons a ℓ p

- Needs to be a mutual definition – cons refers to $\leq_L$, which is indexed by SList.

- Both SList and $\leq_L$ defined inductively – an inductive-inductive definition!
Plan

1. Four slides introduction to Martin-Löf type theory
2. A brief history of inductive types in type theory
3. Inductive-inductive definitions
4. A finite axiomatisation
5. Categorical semantics
Martin-Löf type theory

Five kinds of judgements:

Γ context

Γ ⊢ A : Set

Γ ⊢ r : A

Γ ⊢ A = B : Set

Γ ⊢ r = s : A
Some rules

Forming contexts:

\[ \varepsilon \text{ context} \quad \frac{\Gamma \text{ context} \quad \Gamma \vdash A : \text{Set}}{\Gamma, x : A \text{ context}} \]

Forming types:

\[ \frac{\Gamma \text{ context}}{\Gamma \vdash 1 : \text{Set}} \quad \frac{\Gamma \text{ context} \quad \Gamma \vdash A : \text{Set} \quad \Gamma, x : A \vdash B : \text{Set}}{\Gamma \vdash (\Sigma x : A. B) : \text{Set}} \]

Introducing terms:

\[ \frac{\Gamma \vdash \star : 1}{\Gamma \vdash \star : 1} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b \rangle : \Sigma x : A. B} \]

\[ \vdots \]
Types we will be using

- **Dependent function space** \((x : A) \rightarrow B(x)\) (also written \(\prod_{x : A} B\)).
  - Elements functions \(f\) such that \(f(a) : B(a)\) whenever \(a : A\).
  - Special case: non-dependent function space \(A \rightarrow B\).

- **Dependent pairs** \((x : A) \times B(x)\) (also written \(\Sigma x : A.B\)).
  - Elements pairs \(\langle a, b\rangle\) such that \(a : A\) and \(b : B(a)\).
  - Special case: Cartesian product \(A \times B\).

- **Disjoint union** \(A + B\).
  - Elements \(\text{inl}(a), \text{inr}(b)\) where \(a : A\) and \(b : B\).
  - Can be constructed as \(\Sigma x : 2.\text{if } x \text{ then } A \text{ else } B\) (if large elimination for 2 is available).

- **Empty type** \(0\), **unit type** \(1\) (with inhabitant \(\star : 1\)).

- **Logical Framework formulation of type theory.**
Propositions as types

Propositions can be seen as types:

- Universal quantification \( \forall x \in A. B(x) \) by \((x : A) \to B(x)\).
- Implication \( A \to B \) by \(A \to B\).
- Existential quantification \( \exists x \in A. B(x) \) by \((x : A) \times B(x)\).
- Conjunction \( A \land B \) by \(A \times B\).
- Disjunction \( A \lor B \) by \(A + B\).
- The false proposition \( \bot \) by \(0\) (no proof).
- True propositions by inhabited types.

Will be implicitly used in the rest of the talk.
A brief history of inductive types
In the beginning, there were examples
Martin-Löf (1972, 1979, 1980, . . . )

First accounts of Martin-Löf type theory includes examples of “inductively generated” types:

- \( \mathbb{N} \), finite sets (1972)
- \( W \)-types (1979)
- Kleene’s \( \mathcal{O} \), lists (1980)
- . . .

The system is considered open; new inductive types should be added as needed.

“We can follow the same pattern used to define natural numbers to introduce other inductively defined sets. We see here the example of lists.” – Martin-Löf 1980
Examples of inductive definitions

\[
\begin{align*}
[] & : \text{List}_\mathbb{N} \\
(x : \mathbb{N} \quad xs : \text{List}_\mathbb{N}) & \quad (x :: xs) : \text{List}_\mathbb{N}
\end{align*}
\]

\[
\begin{align*}
0 & : \text{Kleenes}_0 \\
n : \text{Kleenes}_0 & \quad \text{suc}(n) : \text{Kleenes}_0
\end{align*}
\]

\[
\begin{align*}
f & : \mathbb{N} \to \text{Kleenes}_0 \\
\lim(f) & : \text{Kleenes}_0
\end{align*}
\]

\[
\begin{align*}
a & : A \\
f & : B(a) \to W(A, B)
\end{align*}
\]

\[
\begin{align*}
\sup(a, f) & : W(A, B)
\end{align*}
\]

\[
\begin{align*}
data \ \text{List}_\mathbb{N} : \text{Set} \ & \text{where} \\
[] & : \text{List}_\mathbb{N} \\
_: :_ & : \mathbb{N} \to \text{List}_\mathbb{N} \to \text{List}_\mathbb{N}
\end{align*}
\]

\[
\begin{align*}
data \ \text{Kleenes}_0 : \text{Set} \ & \text{where} \\
0 & : \text{Kleenes}_0 \\
S & : \text{Kleenes}_0 \to \text{Kleenes}_0 \\
\lim & : (\mathbb{N} \to \text{Kleenes}_0) \\
& \to \text{Kleenes}_0
\end{align*}
\]

\[
\begin{align*}
data \ W A B : \text{Set} \ & \text{where} \\
\sup & : (a : A) \to \\
(f : B a \to W A B) & \to W A B
\end{align*}
\]
Induction principles/elimination rules

- Each definition has a corresponding induction principle, stating that it is the least set closed under its constructors.

E.g.

\[
\text{elim}_{\text{List}_N} : (P : \text{List}_N \rightarrow \text{Set}) \rightarrow \\
(\text{step}[] : P(\text{[]})) \rightarrow \\
(\text{step}:: : (x : \mathbb{N}) \rightarrow (xs : \text{List}_N) \rightarrow P(xs) \rightarrow P(x :: xs)) \rightarrow \\
(y : \text{List}_N) \rightarrow P(y)
\]

\[
\text{elim}_{\text{List}_N}(P, \text{step}[], \text{step}::, \text{[]}) = \text{step}[] \\
\text{elim}_{\text{List}_N}(P, \text{step}[], \text{step}::, x :: xs) = \text{step}::(x, xs, \text{elim}_{\text{List}_N}(\ldots, xs))
\]

- How can we talk about all inductive definitions?
Church encodings?
Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

- E.g.

\[
\mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X
\]

\[
\text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b)
\]
Church encodings?

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

- E.g.

  \[
  \mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X
  \]

  \[
  \text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b)
  \]

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).
Church encodings?

Pfenning and Paulin-Mohring (1989)

First attempt in Calculus of Constructions: use Church encodings of inductive types.

E.g.

\[ \mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X : \text{Set} \]

\[ \text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b) : \text{Set} \]

Problems:

- Uses impredicativity in an essential way.
- Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).
Church encodings?
Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

- E.g.

  $$\mathbb{N} = \left( X : \text{Set} \right) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X : \text{Set}$$

  $$\text{Id}_A(a, b) = \left( X : A \rightarrow \text{Set} \right) \rightarrow X(a) \rightarrow X(b) : \text{Set}$$

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).
Church encodings?
Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

E.g.

\[ \mathbb{N} = (X : \text{Set}) \to X \to (X \to X) \to X : \text{Set} \]

\[ \text{Id}_A(a, b) = (X : A \to \text{Set}) \to X(a) \to X(b) : \text{Set} \]

Problems:

- Uses impredicativity in an essential way.
- Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).

Solution: Calculus of Inductive Constructions with inductive types builtin (according to schema).
Syntactic schemata
Backhouse (1987), Coquand and Paulin-Mohring (1990), Dybjer (1994), ... 

Dybjer (1994) considers constructors of the form

\[
\text{intro}_U : (A :: \sigma) \\
(b :: \beta[A]) \rightarrow \\
(u :: \gamma[A, b]) \rightarrow \\
U
\]

where

- \( \sigma \) is a sequence of types for parameters \[\text{‘x :: Y’ telescope notation}\]
- \( \beta[A] \) is a sequence of types for non-inductive arguments.
- \( \gamma[A, b] \) is a sequence of types for inductive arguments:
  - Each \( \gamma_i[A, b] \) is of the form \( \xi_i[A, b] \rightarrow U \) (strict positivity).
The elimination and computation rules are determined by an inversion principle.

Infinite axiomatisation.

Inprecise; ‘…’ everywhere.

No way to reason about an arbitrary inductive definition inside the system (generic map etc.).
Syntax internalised


- Setzer wanted to analyse the proof-theoretical strength of Dybjer’s schema version of induction-recursion.

- Hard with lots of ‘...’ around...

- So they developed an axiomatisation where the syntax has been internalised into the system.

- Basic idea (simplified for inductive definitions): the type is “given by constructors”, so describe the domain of the constructor

  \[ \text{intro}_{U_{\gamma}} : \text{Arg}(\gamma, U_{\gamma}) \rightarrow U_{\gamma} \]

  [ \gamma is “code” that contains the necessary information to describe \(U_{\gamma}\).]
Basic idea in some more detail

- Universe SP of codes for the domain of constructors of inductively defined sets. [SP stands for Strictly Positive.]

- Decoding function $\text{Arg} : \text{SP} \to \text{Set} \to \text{Set}$. [$\text{Arg}(\gamma, X)$ is the domain where $X$ is used for the inductive arguments.]

- For every $\gamma : \text{SP}$, stipulate that there is a set $U_{\gamma}$ and a constructor $\text{intro}_{\gamma} : \text{Arg}(\gamma, U_{\gamma}) \to U_{\gamma}$.

- Inversion principle for elimination and computation rules.
SP, Arg and $U_\gamma$

data SP : Set$_1$ where
    nil : SP
    nonind : (A : Set) → (A → SP) → SP
    ind : (A : Set) → SP → SP

Arg : SP → Set → Set
Arg nil X = 1
Arg (nonind A $\gamma$) X = (y : A) × (Arg ($\gamma$ y) X)
Arg (ind A $\gamma$) X = (A → X) × (Arg $\gamma$ X)

data U ($\gamma$ : SP) : Set where
    intro : Arg $\gamma$ (U $\gamma$) → U $\gamma$
Example: the code for $\text{List}_\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda \_ \text{._ind}(1, \text{nil}))$$

with

$\text{List}_\mathbb{N} : \text{Set}$

$\text{List}_\mathbb{N} = U \gamma_{\text{List}_\mathbb{N}}$

$\_ : \text{List}_\mathbb{N}$

$\text{List}_\mathbb{N} \to \text{List}_\mathbb{N}$

$x :: \text{xs} = \{?_1 : \text{List}_\mathbb{N}\}$
Example: the code for $\text{List}_\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda \_. \text{ind}(1, \text{nil}))$$

with

$\text{List}_\mathbb{N} : \text{Set}$

$\text{List}_\mathbb{N} = \text{U } \gamma_{\text{List}_\mathbb{N}}$

$[]): \text{List}_\mathbb{N}$

$[] = \text{intro} \{ ?_2 : \text{Arg}(\gamma_{\text{List}_\mathbb{N}}, \text{List}_\mathbb{N}) \}$

$\_ :: \_ : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N}$

$x :: xs = \{ ?_1 : \text{List}_\mathbb{N} \}$
**Example: the code for List\(_N\)**

We can encode two constructors into one using the dependency on non-inductive arguments:

\[ \gamma +_{SP} \psi := \text{nonind}((2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)) \]

We have

\[ \gamma_{\text{List}_N} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda \ldots . \text{ind}(1, \text{nil})) \]

with

\[ \text{List}_N : \text{Set} \]
\[ \text{List}_N = \bigcup \gamma_{\text{List}_N} \]

\[ \boxed{[]} : \text{List}_N \]
\[ \boxed{[]} = \text{intro} \boxed{\{?_2 : (x : 2) \times (\text{if } x \text{ then } 1 \text{ else } \mathbb{N} \times (1 \to \text{List}_N) \times 1)\}} \]

\[ \boxed{\_ : \_ : \mathbb{N} \to \text{List}_N \to \text{List}_N} \]
\[ \boxed{x :: xs} = \boxed{\{?_1 : \text{List}_N\}} \]
Example: the code for $\text{List}_\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda \cdot \text{ind}(1, \text{nil}))$$

with

$\text{List}_\mathbb{N} : \text{Set}$

$\text{List}_\mathbb{N} = \bigcup \gamma_{\text{List}_\mathbb{N}}$

$[] : \text{List}_\mathbb{N}$

$[] = \text{intro} \langle \{ ?_3 : 2 \}, \{ ?_4 : \text{if } ?_3 \text{ then } 1 \text{ else } \mathbb{N} \times \ldots \} \rangle$

$_{::} : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N}$

$x :: xs = \{ ?_1 : \text{List}_\mathbb{N} \}$
Example: the code for \( \text{List}_\mathbb{N} \)

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +SP \, \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_\mathbb{N}} = \text{nil} +SP \, \text{nonind}(\mathbb{N}, \lambda_. \text{ind}(1, \text{nil}))
\]

with

\( \text{List}_\mathbb{N} : \text{Set} \)

\( \text{List}_\mathbb{N} = \bigcup \gamma_{\text{List}_\mathbb{N}} \)

\( [] : \text{List}_\mathbb{N} \)

\( [] = \text{intro } \langle \text{tt}, \{?4 : 1\} \rangle \)

\( \_ :: \_ : \mathbb{N} \to \text{List}_\mathbb{N} \to \text{List}_\mathbb{N} \)

\( x :: xs = \{?1 : \text{List}_\mathbb{N}\} \)
Example: the code for \(\text{List}_\mathbb{N}\)

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda \_\text{ind}(1, \text{nil}))
\]

with

\[
\text{List}_\mathbb{N} : \text{Set}
\]

\[
\text{List}_\mathbb{N} = \mathbb{U} \gamma_{\text{List}_\mathbb{N}}
\]

\[
[] : \text{List}_\mathbb{N}
\]

\[
[] = \text{intro } \langle \text{tt, } \star \rangle
\]

\[
\_ : \_ : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N}
\]

\[
x :: xs = \{ ?_1 : \text{List}_\mathbb{N} \}
Example: the code for List$^\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}^\mathbb{N}} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda \_ \text{ind}(1, \text{nil}))$$

with

List$^\mathbb{N}$ : Set
List$^\mathbb{N}$ = $\bigcup$ $\gamma_{\text{List}^\mathbb{N}}$

[] : List$^\mathbb{N}$
[] = intro ⟨tt, *⟩

_ :: _ : $\mathbb{N}$ → List$^\mathbb{N}$ → List$^\mathbb{N}$
x :: xs = intro ⟨ff, ${?_5 : \mathbb{N} \times (1 \rightarrow \text{List}^\mathbb{N}) \times 1}$⟩
Example: the code for \( \text{List}_\mathbb{N} \)

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda \text{ind}(1, \text{nil}))
\]

with

\[
\text{List}_\mathbb{N} : \text{Set} \\
\text{List}_\mathbb{N} = \bigcup \gamma_{\text{List}_\mathbb{N}}
\]

\([\ ] : \text{List}_\mathbb{N} \]
\([\ ] = \text{intro} \langle \text{tt}, * \rangle \]

\(\_ : \_ : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N} \)
\(x :: xs = \text{intro} \langle \text{ff}, \langle {?6 : \mathbb{N}} \rangle, \{ {?7 : 1 \rightarrow \text{List}_\mathbb{N}} \}, \{ {?8 : 1} \} \rangle \)
Example: the code for $\text{List}_\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda _. \text{ind}(1, \text{nil}))$$

with

$\text{List}_\mathbb{N} : \text{Set}$

$\text{List}_\mathbb{N} = \mathcal{U} \gamma_{\text{List}_\mathbb{N}}$

$\texttt{[]} : \text{List}_\mathbb{N}$

$\texttt{[]} = \text{intro} \langle \texttt{tt}, * \rangle$

$\texttt{::} : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N}$

$x :: xs = \text{intro} \langle \texttt{ff}, \langle x, \{?7 : 1 \rightarrow \text{List}_\mathbb{N}\}, \{?8 : 1\} \rangle \rangle$
Example: the code for $\text{List}_\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda_. \text{ind}(1, \text{nil}))$$

with

$\text{List}_\mathbb{N} : \text{Set}$

$\text{List}_\mathbb{N} = U \gamma_{\text{List}_\mathbb{N}}$

$[] : \text{List}_\mathbb{N}$

$[] = \text{intro} \langle \text{tt}, * \rangle$

$\_ :: : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N}$

$x :: xs = \text{intro} \langle \text{ff}, \langle x, (\lambda_. xs) \rangle, \{?_8 : 1\} \rangle$
Example: the code for $\text{List}_\mathbb{N}$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(\mathbb{2}, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_\mathbb{N}} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda_. \text{ind}(\mathbb{1}, \text{nil}))$$

with

$\text{List}_\mathbb{N} : \text{Set}$

$\text{List}_\mathbb{N} = \Sigma \gamma_{\text{List}_\mathbb{N}}$

[] : $\text{List}_\mathbb{N}$

[] = intro $\langle \text{tt}, \star \rangle$

$_{::} : \mathbb{N} \rightarrow \text{List}_\mathbb{N} \rightarrow \text{List}_\mathbb{N}$

$x :: xs = \text{intro} \langle \text{ff}, \langle x, (\lambda_.xs), \star \rangle \rangle$
The universe described is very much a low-level construction.

We do not expect the user to deal with the universe directly.

Rather, high-level constructs (data declarations etc) can be translated to a core type theory with a universe of data types.

Makes generic operations (decidable equality, map etc) possible.

Route taken in Epigram 2.


Dagand, McBride: Elaborating Inductive Definitions (2012)
The unstoppable march of progress

- So far, we have described “simple” inductive types.

- When programming or proving with dependent types, one quickly feels the need for more advanced data structures.
  - Inductive families $U : I \to \text{Set}$
  - Induction-recursion $U : \text{Set}, \ T : U \to \text{Set}$
  - Inductive-inductive definitions $A : \text{Set}, \ B : A \to \text{Set}$

- Can we scale the universe just described to handle these data types as well?
The unstoppable march of progress

- So far, we have described “simple” inductive types.

- When programming or proving with dependent types, one quickly feels the need for more advanced data structures.

  - Inductive families $U : I \rightarrow \text{Set}$
  
  - Induction-recursion $U : \text{Set}, \ T : U \rightarrow \text{Set}$
  
  - Inductive-inductive definitions $A : \text{Set}, \ B : A \rightarrow \text{Set}$

- Can we scale the universe just described to handle these data types as well?

- **Anticipated answer:** yes! This talk: inductive-inductive definitions.
Inductive-inductive definitions
What is an inductive-inductive definition?

- Induction-induction is a principle for defining data types $A : \text{Set}$, $B : A \rightarrow \text{Set}$.

- Both $A$ and $B$ are defined inductively, “given by constructors”.

What is an inductive-inductive definition?

- Induction-induction is a principle for defining data types $A : \text{Set}$, $B : A \rightarrow \text{Set}$.

- Both $A$ and $B$ are defined inductively, “given by constructors”.

- $A$ and $B$ are defined simultaneously, so the constructors for $A$ can refer to $B$ and vice versa.

- In addition, the constructors for $B$ can even refer to the constructors for $A$. 
Induction versus recursion

- I mean induction as a definitional principle.

- “All natural numbers are generated from zero and successor.”

- By recursion, I mean a structured way to take apart something which is defined by induction.

- “Plus is defined by recursion on its first argument.”

- Important to see the difference between induction-recursion and induction-induction.
Induction versus recursion

- I mean induction as a definitional principle.

- “All natural numbers are generated from zero and successor.”

- By recursion, I mean a structured way to take apart something which is defined by induction.

- “Plus is defined by recursion on its first argument.”

- Important to see the difference between induction-recursion and induction-induction.

- Proof by induction is just dependent recursion.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: $\mathbb{N}$)
   - Because we define $A : \text{Set}$ and $B : A \rightarrow \text{Set}$ simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \rightarrow \text{Set}$ is indexed by $A$.

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set $A : \text{Set}$ is defined along with $B : A \rightarrow \text{Set}$, and not fixed beforehand.

4. An inductive-recursive definition (example: universes in type theory)
   - Because $B : A \rightarrow \text{Set}$ is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However, 4 is not.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: \( \mathbb{N} \))
   - Because we define \( A : \text{Set} \) and \( B : A \to \text{Set} \) simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because \( B : A \to \text{Set} \) is indexed by \( A \).

However, conjecture that it can be reduced to IID.

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set \( A : \text{Set} \) is defined along with \( B : A \to \text{Set} \), and not fixed beforehand.

4. An inductive-recursive definition (example: universes in type theory)
   - Because \( B : A \to \text{Set} \) is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However 4 is not.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: $\mathbb{N}$)
   - Because we define $A : \text{Set}$ and $B : A \rightarrow \text{Set}$ simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \rightarrow \text{Set}$ is indexed by $A$.

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set $A : \text{Set}$ is defined along with $B : A \rightarrow \text{Set}$, and not fixed beforehand.
   - However, conjecture that it can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because $B : A \rightarrow \text{Set}$ is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However 4 is not.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: \( \mathbb{N} \))
   - Because we define \( A : \text{Set} \) and \( B : A \rightarrow \text{Set} \) simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because \( B : A \rightarrow \text{Set} \) is indexed by \( A \).

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set \( A : \text{Set} \) is defined along with \( B : A \rightarrow \text{Set} \), and not fixed beforehand.
   - However, conjecture that it can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because \( B : A \rightarrow \text{Set} \) is defined inductively, not recursively.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: \( \mathbb{N} \))
   - Because we define \( A : \text{Set} \) and \( B : A \rightarrow \text{Set} \) simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because \( B : A \rightarrow \text{Set} \) is indexed by \( A \).

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set \( A : \text{Set} \) is defined along with \( B : A \rightarrow \text{Set} \), and not fixed beforehand.
   - However, conjecture that it can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because \( B : A \rightarrow \text{Set} \) is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However 4 is not.
Examples of inductive-inductive definitions
Instances of induction-induction have been used implicitly by

- **Dybjer** (Internal type theory, 1996),
- **Danielsson** (A formalisation of a dependently typed language as an inductive-recursive family, 2007), and
- **Chapman** (Type theory should eat itself, 2009)

to model dependent type theory inside itself.
Type theory inside type theory

- $\text{Ctxt} : \text{Set}$
- $\text{Ty} : \text{Ctxt} \rightarrow \text{Set}$
- $\text{Term} : (\Gamma : \text{Ctxt}) \rightarrow \text{Ty}(\Gamma) \rightarrow \text{Set}$
- ... Substitutions, ...
- ...
The crucial point

- The empty context $\varepsilon$ is a well-formed context.

$\varepsilon : \text{Ctx}$
The crucial point

- The empty context $\varepsilon$ is a well-formed context.
- If $\tau$ is a well-formed type in context $\Gamma$, then $\Gamma, x : \tau$ is a well-formed context.

\[
\varepsilon : \text{Ctxt}
\]

\[
\Gamma : \text{Ctxt} \quad \tau : \text{Ty}(\Gamma) \\
\Gamma \triangleright \tau : \text{Ctxt}
\]
Constructor for Ty referring to constructor for Ctxt

\[
\begin{align*}
\Gamma \text{ context} & \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
\Gamma & \vdash \sum x : \sigma . \tau(x) \text{ type}
\end{align*}
\]
Constructor for Ty referring to constructor for Ctxt

\[ \Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \]
\[ \Gamma \vdash \sum x : \sigma \cdot \tau(x) \text{ type} \]

\[ \Gamma : \text{Ctxt} \]

(Also have base type \( \iota \) in any context: \( \Gamma : \text{Ctxt} \quad \iota \Gamma : \text{Ty}(\Gamma) \))
Constructor for Ty referring to constructor for Ctxt

\[ \Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \]
\[ \Gamma \vdash \sum x : \sigma . \tau(x) \text{ type} \]

\[ \Gamma : \text{Ctxt} \quad \sigma : \text{Ty}(\Gamma) \]
Constructor for Ty referring to constructor for Ctxt

\[ \Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \]

\[ \Gamma \vdash \Sigma x : \sigma . \tau(x) \text{ type} \]

\[ \Gamma : \text{Ctxt} \quad \sigma : \text{Ty}(\Gamma) \quad \tau : \text{Ty}(\Gamma \triangleright \sigma) \]
Constructor for Ty referring to constructor for Ctxt

\[
\Gamma \text{ context } \quad \Gamma \vdash \sigma \text{ type } \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type } \\
\quad \Gamma \vdash \Sigma x : \sigma . \tau(x) \text{ type }
\]

\[
\Gamma : \text{Ctxt} \quad \sigma : \text{Ty}(\Gamma) \quad \tau : \text{Ty}(\Gamma \triangleright \sigma)
\]
Constructor for Ty referring to constructor for Ctxt

\[
\begin{align*}
\Gamma \text{ context} & \quad \Gamma \vdash \sigma \text{ type} & \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
& \quad \Gamma \vdash \sum_{x : \sigma} . \tau(x) \text{ type}
\end{align*}
\]

\[
\begin{align*}
\Gamma : \text{Ctxt} & \quad \sigma : \text{Ty}(\Gamma) & \quad \tau : \text{Ty}(\Gamma \uplus \sigma) \\
& \quad \Sigma(\sigma, \tau) : \text{Ty}(\Gamma)
\end{align*}
\]
Constructor for $\text{Ty}$ referring to constructor for $\text{Ctxt}$

\[
\begin{array}{c}
\Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
\hline
\Gamma \vdash \sum x : \sigma . \tau(x) \text{ type}
\end{array}
\]

\[
\begin{array}{c}
\Gamma : \text{Ctxt} \quad \sigma : \text{Ty}(\Gamma) \quad \tau : \text{Ty}(\Gamma \triangleright \sigma) \\
\hline
\sum (\sigma, \tau) : \text{Ty}(\Gamma)
\end{array}
\]

(Also have base type $\iota$ in any context:

\[
\begin{array}{c}
\Gamma : \text{Ctxt} \\
\hline
\iota_{\Gamma} : \text{Ty}(\Gamma)
\end{array}
\]
Conway’s surreal numbers

- Totally ordered Field containing the reals and the ordinals (at least classically).

- “Fills the holes” between them as well (think infinitesimals).

- Constructed in one step, instead of $\mathbb{N} \leadsto \mathbb{Z} \leadsto \mathbb{Q} \leadsto \mathbb{R}$.

- John Conway: *On Numbers and Games*.

- Donald Knuth: *Surreal Numbers*.
Definition (Dedekind cut)

A Dedekind cut \((L, R)\) consists of two non-empty sets of rational numbers \(L, R \subseteq \mathbb{Q}\) such that

- \(L \cup R = \mathbb{Q}\),
- All elements of \(L\) are less than all elements of \(R\),
- \(L\) contains no greatest element.
Definition (Surreal number)

A surreal number \((L, R)\) consists of two non-empty sets of rational numbers \(L, R \subseteq \mathbb{Q}\) such that

- \(L \cup R = \mathbb{Q}\),
- All elements of \(L\) are less than all elements of \(R\),
- \(L\) contains no greatest element.
Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of rational numbers \( L, R \subseteq \mathbb{Q} \) such that

- \( L \cup R = \mathbb{Q} \),
- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \( L \cup R = \mathbb{Q} \),
- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)
A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \( L \cup R = \mathbb{Q} \),
- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \( (\forall x^L \in L)(\forall x^R \in R) \neg(x^L \geq x^R) \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{ L \mid R \} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L) (\forall x^R \in R) \neg (x^L \geq x^R),
\]

- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R),
\]

\( L \) contains no greatest element.
Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R).
\]

All surreal numbers are constructed this way.
From Dedekind cuts to surreal numbers

Definition

A surreal number \( \{ L \| R \} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R).
\]

All surreal numbers are constructed this way.
Definition

A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg(x^L \geq x^R).
\]

All surreal numbers are constructed this way.

Definition

Let \( x = \{X_L|X_R\}, \ y = \{Y_L|Y_R\} \). We say \( x \geq y \) iff

\[
(\forall x^R \in X_R) \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L) \neg(y^L \geq x)
\]
From Dedekind cuts to surreal numbers

Definition

A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg(x^L \geq x^R).
\]

All surreal numbers are constructed this way.

Definition

Let \( x = \{X_L|X_R\}, y = \{Y_L|Y_R\} \). We say \( x \geq y \) iff

\[
(\forall x^R \in X_R) \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L) \neg(y^L \geq x)
\]

An inductive-inductive definition!
An inductive-inductive definition

Define simultaneously

\[
\text{Surreal} : \text{Set} \\
\leq : \text{Surreal} \to \text{Surreal} \to \text{Set} \\
\not\leq : \text{Surreal} \to \text{Surreal} \to \text{Set}
\]

Need to encode some set theory such as \( \mathcal{P}(\text{Surreal}) \) and \( x \in X_L \) in type theory – we deal with this informally.

(Use \( \mathcal{P}(X) := \Sigma a : U. T(a) \to X \) for some universe \((U, T)\). See e.g. Aczel’s interpretation of CZF in type theory (Aczel 1978).)
A surreal number \( \{X_L|X_R\} \) consists of two sets of surreal numbers \( X_L, X_R \) such that

\[
(\forall x^L \in X_L)(\forall x^R \in X_R) \neg(x^L \geq x^R).
\]

All surreal numbers are constructed this way.

**data** Surreal : Set where

\[
\begin{align*}
\{\_|\_\} : (X_L : \mathcal{P}(\text{Surreal})) & \rightarrow (X_R : \mathcal{P}(\text{Surreal})) \\
& \rightarrow (\forall x^L \in X_L)(\forall x^R \in X_R)((x^L \geq x^R) \rightarrow \bot) \\
& \rightarrow \text{Surreal}
\end{align*}
\]
Constructor for Surreal

Definition

A surreal number \( \{X_L | X_R\} \) consists of two sets of surreal numbers \( X_L, X_R \) such that

\[
(\forall x^L \in X_L)(\forall x^R \in X_R) \neg(x^L \geq x^R).
\]

All surreal numbers are constructed this way.

**data** Surreal : Set where

\[
\{ - | - \} : (X_L : \mathcal{P}(\text{Surreal})) \rightarrow (X_R : \mathcal{P}(\text{Surreal}))
\]

\[
\rightarrow (\forall x^L \in X_L)(\forall x^R \in X_R)((x^L \geq x^R) \rightarrow \bot)
\]

\[
\rightarrow \text{Surreal}
\]
Constructor for Surreal

Definition

A surreal number \( \{X_L|X_R\} \) consists of two sets of surreal numbers \( X_L, X_R \) such that

\( \forall x^L \in X_L \)(\( \forall x^R \in X_R \))\( \neg(x^L \geq x^R) \).

All surreal numbers are constructed this way.

**data** Surreal

\( \{-\} : (X_L : P(Surreal)) \land (X_R : P(Surreal)) \rightarrow (\forall x^L \in X_L)(\forall x^R \in X_R)((x^L \geq x^R) \rightarrow \bot) \)

\rightarrow Surreal

We cannot have negative occurrences of the set we are defining!
Constructor for Surreal

Definition

A surreal number \( \{X_L|X_R\} \) consists of two sets of surreal numbers \( X_L, X_R \) such that

\[
(\forall x^L \in X_L)(\forall x^R \in X_R) \neg(x^L \geq x^R).
\]

All surreal numbers are constructed this way.

\[
\text{data Surreal : Set where}
\]

\[
\text{\{-\} : (X_L : P(Surreal)) \rightarrow (X_R : P(Surreal))}
\]

\[
\rightarrow (\forall x^L \in X_L)(\forall x^R \in X_R)(x^L \not\geq x^R)
\]

\[
\rightarrow \text{Surreal}
\]
Negative occurrences of $\geq$

**Definition**

Let $x = \{X_L|X_R\}$, $y = \{Y_L|Y_R\}$. We say $x \geq y$ iff

$$(\forall x^R \in X_R) \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L) \neg(y^L \geq x)$$

- Define $x \geq y$ and $x \not\geq y$ simultaneously.

- $\neg(x \geq y)$ iff

  $$\neg(((\forall x^R \in X_R) \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L) \neg(y^L \geq x)))$$

  if

  $$(\exists x^R \in X_R)(y \geq x^R) \text{ or } (\exists y^L \in Y_L)(y^L \geq x)$$

  (also “only if” with classical logic).

- So we define $x \not\geq y$ iff

  $$(\exists x^R \in X_R)(y \geq x^R) \text{ or } (\exists y^L \in Y_L)(y^L \geq x)$$
Mutual definition of $\geq$ and $\not\geq$

**Definition**

Let $x = \{X_L|X_R\}$, $y = \{Y_L|Y_R\}$. We say $x \geq y$ iff

$$(\forall x^R \in X_R) \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L) \neg(y^L \geq x)$$

---

**data** $\geq$: Surreal $\rightarrow$ Surreal $\rightarrow$ Set where

$$geq : \ldots X_L, X_R, p \ldots$$

$$\rightarrow \ldots Y_L, Y_R, q \ldots$$

$$\rightarrow (\forall x^R \in X_R)(\{Y_L|Y_R\}_q \not\geq x^R)$$

$$\rightarrow (\forall y^L \in Y_L)(y^L \not\geq \{X_L|X_R\}_p)$$

$$\rightarrow \{X_L|X_R\}_p \geq \{Y_L|Y_R\}_q$$
Mutual definition of $\geq$ and $\not\geq$ (cont.)

$\neg (x \geq y)$ if

$$(\exists x^R \in X_R) (y \geq x^R) \text{ or } (\exists y^L \in Y_L) (y^L \geq x)$$

data $\not\geq : \text{Surreal} \rightarrow \text{Surreal} \rightarrow \text{Set}$ where

$$ngeql : \ldots X_L, X_R, p \ldots$$

$$\rightarrow \ldots Y_L, Y_R, q \ldots$$

$$\rightarrow (\exists x^R \in X_R)(\{ Y_L | Y_R \}_q \geq x^R)$$

$$\rightarrow \{ X_L | X_R \}_p \not\geq \{ Y_L | Y_R \}_q$$

$$ngeqr : \ldots X_L, X_R, p \ldots$$

$$\rightarrow \ldots Y_L, Y_R, q \ldots$$

$$\rightarrow (\exists y^L \in Y_L)(y^L \geq \{ X_L | X_R \}_p)$$

$$\rightarrow \{ X_L | X_R \}_p \not\geq \{ Y_L | Y_R \}_q$$
Constructing the Field structure

- Can then use the elimination rules for inductive-inductive definitions to define negation, addition, multiplication . . .

- Typical pattern: need to define the operation and prove that it preserves the order structure etc simultaneously.

- Work in progress.

- **Related work:** Mamane: Surreal Numbers in Coq (2006)
  - Encoding of surreal numbers, since Coq does not support induction-induction.
A finite axiomatisation
An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?
An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?
- High-level idea: Add a universe (family) \( SP = (SP^0_A, SP^0_B) \) of codes representing the inductive-inductively defined sets.
An axiomatisation

How to axiomatise a type theory with inductive-inductive definitions?

High-level idea: Add a universe (family) $\text{SP} = (\text{SP}_A^0, \text{SP}_B^0)$ of codes representing the inductive-inductively defined sets.

Stipulate that for each code $\gamma = (\gamma_A, \gamma_B)$, there are

$$A_\gamma : \text{Set}$$
$$B_\gamma : A_\gamma \rightarrow \text{Set}$$

and constructors

$$\text{intro}_A : \text{Arg}^0_A(\gamma_A, A_\gamma, B_\gamma) \rightarrow A_\gamma$$
$$\text{intro}_B : (x : \text{Arg}^0_B(\gamma_B, A_\gamma, B_\gamma, \text{intro}_A)) \rightarrow B_\gamma(i_\gamma(x))$$
An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?

- High-level idea: Add a universe (family) $SP = (SP^0_A, SP^0_B)$ of codes representing the inductive-inductively defined sets.

- Stipulate that for each code $\gamma = (\gamma_A, \gamma_B)$, there are

  $$A_\gamma : \text{Set}$$
  $$B_\gamma : A_\gamma \to \text{Set}$$

  and constructors

  $$\text{intro}_A : \text{Arg}_A^0(\gamma_A, A_\gamma, B_\gamma) \to A_\gamma$$
  $$\text{intro}_B : (x : \text{Arg}_B^0(\gamma_B, A_\gamma, B_\gamma, \text{intro}_A)) \to B_\gamma(i_\gamma(x))$$

- The codes describe the “pattern functors” $\text{Arg}_A^0, \text{Arg}_B^0$. 
Main idea

- We define
  - a set
    \[ \text{SP}_A^0 : \text{Set} \]
    of codes for inductive definitions for \( A \),
  - a set
    \[ \text{SP}_B^0 : \text{SP}_A^0 \rightarrow \text{Set} \]
    of codes for inductive definitions for \( B \).
  - the set of arguments for the constructor of \( A \):
    \[ \text{Arg}_A^0 : \text{SP}_A^0 \rightarrow (X : \text{Set}) \rightarrow (Y : X \rightarrow \text{Set}) \rightarrow \text{Set} \]
the set of arguments and indices for the constructor of $B$:

$$
\text{Arg}_B^0 : (\gamma_A : SP_A^0) \to \\
(\gamma_B : SP_B^0(\gamma_A)) \\
(X : \text{Set}) \to \\
(Y : X \to \text{Set}) \to \\
(\text{intro}_A : \text{Arg}_A^0(\gamma_A, X, Y) \to X) \\
\to \text{Set}
$$

Index$_B^0 : \cdots$ same arguments as Arg$_B^0 \cdots$

$$
\text{Arg}_B^0(\gamma_A, \gamma_B, X, Y, \text{intro}_A) \to X
$$
Formation and introduction rules

Formation rules:

\[ A_{\gamma_A, \gamma_B} : \text{Set} \quad B_{\gamma_A, \gamma_B} : A_{\gamma_A, \gamma_B} \rightarrow \text{Set} \]

Introduction rule for \( A_{\gamma_A, \gamma_B} \):

\[
\begin{align*}
\text{let } a & : \text{Arg}^0_A(\gamma_A, A_{\gamma_A, \gamma_B}, B_{\gamma_A, \gamma_B}) \\
\text{then } \text{intro}_{A_{\gamma_A, \gamma_B}}(a) & : A_{\gamma_A, \gamma_B}
\end{align*}
\]

Introduction rule for \( B_{\gamma_A, \gamma_B} \):

\[
\begin{align*}
\text{let } a & : \text{Arg}^0_B(\gamma_A, \gamma_B, A_{\gamma_A, \gamma_B}, B_{\gamma_A, \gamma_B}, \text{intro}_{A_{\gamma_A, \gamma_B}}) \\
\text{then } \text{intro}_{B_{\gamma_A, \gamma_B}}(a) & : B_{\gamma_A, \gamma_B}(\text{Index}^0_B(\gamma_A, \gamma_B, A_{\gamma_A, \gamma_B}, B_{\gamma_A, \gamma_B}, \text{intro}_{A_{\gamma_A, \gamma_B}}, a))
\end{align*}
\]

Elimination rules: no problem in extensional type theory, not so easy intentionally.
Definition of $SP_A$

- Instead of defining $SP_A^0$ we define a more general set

$$SP_A : (X_{\text{ref}} : \text{Set}) \rightarrow \text{Set}$$

with a set $X_{\text{ref}}$ of elements of the set to be defined which we can refer to.

- In definition of $\text{Arg}_A$, also require function

$$\text{rep}_X : X_{\text{ref}} \rightarrow X$$

mapping elements in $X_{\text{ref}}$ to the element in $X$ they represent.

- Then

$$SP_A^0 := SP_A(0)$$
$$\text{rep}_X = !_X : 0 \rightarrow X$$
The codes in $\text{SP}_A$

nil

Base case; $\text{intro}_A : 1 \rightarrow A$.

\begin{align*}
\text{nil} : \text{SP}_A(X_{\text{ref}}) \\
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1
\end{align*}
The codes in $\text{SP}_A$

non-ind

Noninductive argument; $\text{intro}_A : ((x : K) \times \ldots) \rightarrow A$.

\[
\frac{K : \text{Set} \quad \gamma : K \rightarrow \text{SP}_A(X_{\text{ref}})}{
\text{non-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}})}
\]

$\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) = 1$
The codes in $\mathcal{SP}_A$

non-ind

Noninductive argument; $\text{intro}_A : ((x : K) \times \ldots) \rightarrow A.$

$K : \text{Set} \quad \gamma : K \rightarrow \mathcal{SP}_A(X_{\text{ref}}) \quad \frac{}{\text{non-ind}(K, \gamma) : \mathcal{SP}_A(X_{\text{ref}})}$

$\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) = 1$

$\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) = (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X)$
The codes in $\text{SP}_A$

A-ind

Inductive argument in $A$; $\text{intro}_A : ((g : K \to A) \times \ldots) \to A$.

\[
\begin{align*}
K : \text{Set} & \quad \gamma : \text{SP}_A(X_{\text{ref}} + K) \\
\text{A-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}}) & \quad \in \text{later arguments, we can refer to} \\
\end{align*}
\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) & = 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) & = \\
& (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X)
\end{align*}
\]
The codes in $\text{SP}_A$

A-ind

Inductive argument in $A$; $\text{intro}_A : (\langle g : K \to A \rangle \times \ldots) \to A$.

\[
\begin{align*}
K : \text{Set} & \quad \gamma : \text{SP}_A(X_{\text{ref}} + K) \\
\text{A-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}})
\end{align*}
\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) &= (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) &= (g : K \to X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g])
\end{align*}
\]
The codes in $\text{SP}_A$

**A-ind**

Inductive argument in $A$; $\text{intro}_A : ((g : K \rightarrow A) \times \ldots) \rightarrow A$.

\[
\begin{align*}
K : \text{Set} & \quad \gamma : \text{SP}_A(X_{\text{ref}} + K) \\
\text{A-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}})
\end{align*}
\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) &= (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) &= (g : K \rightarrow X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g])
\end{align*}
\]

In later arguments, we can refer to

\[
X_{\text{ref}} \cup \{g(x) | x \in K\} \subseteq X,
\]

represented by $[\text{rep}_X, g] : X_{\text{ref}} + K \rightarrow X$. 

The codes in $\text{SP}_A$

**B-ind**

Inductive argument in $B$; $\text{intro}_A : \left( (g : (x : K) \to B(i(x))) \times \ldots \right) \to A$.

$$
\begin{array}{c}
K : \text{Set} \\
h_{\text{index}} : K \to X_{\text{ref}} \\
\gamma : \text{SP}_A
\end{array}
\quad
\frac{
\text{B-ind}(K, h_{\text{index}}, \gamma) : \text{SP}_A
}{
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) = 1
}
\quad
\frac{
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) =
}{
(x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X)
}
\quad
\frac{
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) =
}{
(g : K \to X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g])
}\end{array}
$$
The codes in $\text{SP}_A$

**B-ind**

Inductive argument in $B$; $\text{intro}_A : ( (g : (x : K) \rightarrow B(i(x))) \times \ldots ) \rightarrow A$. 

\[
\begin{align*}
K : \text{Set} & \quad h_{\text{index}} : K \rightarrow X_{\text{ref}} & \quad \gamma : \text{SP}_A \\
\text{B-ind}(K, h_{\text{index}}, \gamma) : \text{SP}_A
\end{align*}
\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) &= (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) &= (g : K \rightarrow X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g]) \\
\text{Arg}_A(X_{\text{ref}}, \text{B-ind}(K, h_{\text{index}}, \gamma), X, Y, \text{rep}_X) &= (g : (x : K) \rightarrow Y((\text{rep}_X \circ h_{\text{index}})(x))) \times \text{Arg}_A(X_{\text{ref}}, \gamma, X, Y, \text{rep}_X)
\end{align*}
\]
An example

The constructor

\[ \triangleright : ((\Gamma : \text{Ctxt}) \times \text{Ty}(\Gamma)) \rightarrow \text{Ctxt} \]

is represented by the code

\[ \gamma\triangleright = \text{A-ind}(1, \text{B-ind}(1, \lambda(\star : 1) \cdot \text{inr}(\star), \text{nil})) \]

We have

\[ \text{Arg}_A(0, \gamma\triangleright, \text{Ctxt}, \text{Ty}, !_{\text{Ctxt}}) = (\Gamma : 1 \rightarrow \text{Ctxt}) \times (1 \rightarrow \text{Ty}(\Gamma(\star))) \times 1 \]

\[ \cong (\Gamma : \text{Ctxt}) \times \text{Ty}(\Gamma) \]
The codes in $\text{SP}_B$

- The universe $\text{SP}_B^0 : \text{SP}_A^0 \to \text{Set}$ is similar to $\text{SP}_A^0$.

- Need argument $\text{SP}_A^0$ to know the shape of constructor for the first set, which can appear in indices.

- We omit the definition here.
Categorical semantics
Initial-algebra like semantics

Joint work with Thorsten Altenkirch and Peter Morris (CALCO 2011)

- Thorsten was not happy with the axiomatisation presented.
- He wanted something cleaner, like initial-algebra semantics.
- However, seem to need to use dialgebras \( f : F(A) \to G(A) \) instead of ordinary algebras \( f : F(A) \to A \).
Dialgebras

**Definition**

Let $F, G : \mathbb{C} \to \mathbb{D}$ be functors. An $(F, G)$-dialgebra $(X, f)$ consists of $X \in \mathbb{C}$ and $f : F(X) \to G(X)$. A morphism between dialgebras $(X, f)$ and $(Y, g)$ is a morphism $\alpha : X \to Y$ in $\mathbb{C}$ such that

\[
\begin{array}{ccc}
F(X) & \xrightarrow{f} & G(X) \\
\downarrow F(\alpha) & & \downarrow G(\alpha) \\
F(Y) & \xrightarrow{g} & G(Y)
\end{array}
\]

Write $\text{Dialg}(F, G)$ for the category of $(F, G)$-dialgebras.

Of course, $G = \text{id} : \mathbb{C} \to \mathbb{C}$ gives ordinary $F$-algebras as a special case.
**Arg\textsubscript{A} and Arg\textsubscript{B} as functors**

**Theorem (extensional type theory)**

For all \( \gamma\textsubscript{A}, \gamma\textsubscript{B}, \text{Arg}\textsubscript{A}(\gamma\textsubscript{A}) \) and \( \text{Arg}\textsubscript{B}(\gamma\textsubscript{A}, \gamma\textsubscript{B}) \) extends to functors

\[
\text{Arg}\textsubscript{A}(\gamma\textsubscript{A}) : \text{Fam}(\text{Set}) \rightarrow \text{Set}
\]

\[
\text{Arg}\textsubscript{B}(\gamma\textsubscript{A}, \gamma\textsubscript{B}) : \text{Dialg}(\text{Arg}\textsubscript{A}(\gamma\textsubscript{A}), \pi\textsubscript{0}) \rightarrow \text{Fam}(\text{Set})
\]

where \( \pi\textsubscript{0} : \text{Fam}(\text{Set}) \rightarrow \text{Set} \) is defined by \( \pi\textsubscript{0}(A, B) = A \).

**Definition of \( \mathbb{E}_{\gamma\textsubscript{A}, \gamma\textsubscript{B}} \)**

Using a pullback of categories, one can define a subcategory \( \mathbb{E}_{\gamma\textsubscript{A}, \gamma\textsubscript{B}} \) of the category \( \text{Dialg}(\text{Arg}\textsubscript{B}, V) \) playing the role of the category of algebras.

\( V : \text{Dialg}(\text{Arg}\textsubscript{A}, U) \rightarrow \text{Fam}(\text{Set}) \) is the forgetful functor \( V(X, f) = X \).
Elimination rules from initiality

One can then show:

**Theorem (extensional type theory)**

*For an inductive-inductive definition given by a code $(\gamma_A, \gamma_B)$, the elimination rules hold if and only if $E_{\gamma_A, \gamma_B}$ has an initial object.*

Main obstacle: Initiality gives non-dependent functions, elimination rules dependent. **Solution:** Use $\Sigma$-types.
Concluding remarks
Not supported in Coq or Epigram.

Is supported in Agda!

Now we know it is sound as well...
Conjecture: reducible to indexed inductive definitions

- It seems as if the theory of inductive-inductive definitions can be reduced to the (extensional) theory of indexed inductive definitions.

- Define simultaneously

\[ A_{\text{pre}} : \text{Set} \quad B_{\text{pre}} : \text{Set} \]

ignoring dependencies of \( B \) on \( A \).

- Then select \( A \subseteq A_{\text{pre}}, \ B \subseteq B_{\text{pre}} \) that satisfy the typing by two inductively defined predicates (indexed inductive definitions).

- Implicitly used by Conway (and Mamane) for the surreal numbers (games).
Summary

Take away message 1
When programming with dependent types, one naturally wants more advanced data structures such as inductive-inductive definitions.

Take away message 2
By using a universe of data types, they can be internalised into the type theory, useful e.g. for generic programming.

- Will hopefully turn into a thesis in the spring.
Take away message 1
When programming with dependent types, one naturally wants more advanced data structures.

Take away message 2
By using a universe of data types, they can be internalised into the type theory, useful e.g. for generic programming.

- Will hopefully turn into a thesis in the spring.